On Inter-deriving Small-step and Big-step Semantics: A Case Study for Storeless Call-by-need Evaluation

Olivier Danvy[a,*], Kevin Millikin[b], Johan Munk[c], Ian Zerny[a,1,*]

[a] *Dept. of Computer Science, Aarhus University, Aabogade 34, DK-8200 Aarhus N*
[b] *Google, Aabogade 15, DK-8200 Aarhus N*
[c] *Arctic Lake Systems, Aabogade 15, DK-8200 Aarhus N*

Abstract

Starting from the standard call-by-need reduction for the λ-calculus that is common to Ariola, Felleisen, Maraist, Odersky, and Wadler, we inter-derive a series of hygienic semantic artifacts: a reduction-free storeless abstract machine, a continuation-passing evaluation function, and what appears to be the first heapless natural semantics for call-by-need evaluation. Furthermore we observe that the evaluation function implementing this natural semantics is in defunctionalized form. The refunctionalized counterpart of this evaluation function implements an extended direct semantics in the sense of Cartwright and Felleisen.

Overall, the semantic artifacts presented here are simpler than many other such artifacts that have been independently worked out, and which require ingenuity, skill, and independent soundness proofs on a case-by-case basis. They are also simpler to inter-derive because the inter-derivational tools (e.g., refocusing and defunctionalization) already exist.

List of Figures

1 Recomposition of outside-in contexts . . . 11
2 Recomposition of inside-out contexts . . . 11
3 Decomposition of an answer term into itself and of a non-answer term into a potential redex and its evaluation context 12
4 Reduction-based refocusing . . . 17
5 Reduction-free refocusing . . . 17
6 Storeless abstract machine for call-by-need evaluation . . . 18
7 The storeless abstract machine of Figure 6 after transition compression . . 20
8 Heapless natural semantics for call-by-need evaluation . . . 23
9 The heapless natural semantics of Figure 8 after refunctionalization 24

*Corresponding authors
Email addresses: `danvy@cs.au.dk` (Olivier Danvy), `kmillikin@google.com` (Kevin Millikin), `johanmunk@gmail.com` (Johan Munk), `zerny@cs.au.dk` (Ian Zerny)

[1]Ian Zerny is a recipient of the Google Europe Fellowship in Programming Technology, and this research is supported in part by this Google Fellowship.

Contents

1 Introduction **3**

2 The standard call-by-name reduction for the λ-calculus **4**

3 The standard call-by-need reduction for the λ-calculus **6**

4 Some exegesis **8**
- 4.1 Potential redexes . . . 9
- 4.2 Barendregt's variable convention . . . 9
- 4.3 The evaluation contexts . . . 10
- 4.4 Recomposition . . . 11
- 4.5 Decomposition . . . 12
- 4.6 The contraction rules . . . 13
- 4.7 Standard one-step reduction . . . 14
- 4.8 Standard reduction-based evaluation . . . 15
- 4.9 Conclusion and perspectives . . . 15

5 From reduction semantics to abstract machine **16**
- 5.1 Refocusing: from reduction semantics to abstract machine . . . 16
- 5.2 Transition compression: from abstract machine to abstract machine . . . 18

6 Small-step abstract machines define relations, big-step abstract machines define functions **20**

7 From abstract machine to evaluation functions **21**
- 7.1 Refunctionalization: from abstract machine to continuation-passing interpreter . . . 21
- 7.2 Back to direct style: from continuation-passing interpreter to natural semantics . . . 22
- 7.3 Refunctionalization: from natural semantics to higher-order evaluation function . . . 24

8 Deterministic abstract machines define functions **25**

9 Conclusion **26**

Appendix A On refunctionalizing and going back to direct style **26**
- Appendix A.1 Abstract machine for evaluating arithmetic expressions . . . 26
- Appendix A.2 Small-step implementation of the abstract machine . . . 27
- Appendix A.3 Big-step implementation of the abstract machine . . . 28
- Appendix A.4 Continuation-passing evaluator . . . 28
- Appendix A.5 Direct-style evaluator . . . 29
- Appendix A.6 Natural semantics . . . 29

Appendix B On the control pattern underlying call by need 29
Appendix B.1 Function-based encoding 30
Appendix B.2 Continuation-based encoding 30
Appendix B.3 State-based encoding 31

1. Introduction

A famous functional programmer once was asked to give an overview talk. He began with "This talk is about lazy functional programming and call by need." and paused. Then, quizzically looking at the audience, he quipped: "Are there any questions?" There were some, and so he continued: "Now listen very carefully, I shall say this only once."

This apocryphal story illustrates *demand-driven computation* and *memoization of intermediate results*, two key features that have elicited a fascinating variety of semantic specifications and implementation techniques over the years, ranging from purely syntactic treatments to mutable state, and featuring small-step operational semantics [1, 2], a range of abstract machines [3–5], big-step operational semantics [6, 7], as well as evaluation functions [8, 9].

In this article, we extract the computational content of the standard call-by-need reduction for the λ-calculus that is common to both Ariola and Felleisen [1] and Maraist, Odersky, and Wadler [2]. This computational content takes the forms of a one-step reduction relation, an abstract machine, and a natural semantics that are mutually compatible and all abide by Barendregt's variable convention [10, page 26]. Traditionally, one could either handcraft each of these semantic artifacts from scratch and then prove a series of soundness theorems, or invent a calculation to go from artifact to artifact and prove the correctness of the calculation on the way. We depart from these two traditions by going from artifact to artifact using a pre-defined series of fully correct transformations, following the programme outlined in the first author's invited talk at ICFP 2008 [11]. To this programme, though, we add one new refunctionalization step that is specific to call by need. The inter-derivation is itemized as follows:

0. for starters, we make the contraction rules explicitly hygienic to make the standard one-step reduction preserve Barendregt's variable convention;
1. iterating this hygienic standard one-step reduction yields a standard reduction-based evaluation, which we refocus [12] to obtain a reduction-free evaluation with the same built-in hygiene; this reduction-free evaluation takes the form of an abstract machine and is correct by construction; we simplify this hygienic abstract machine by hereditarily compressing its corridor transitions.

We then change perspective and instead of considering this abstract machine as a small-step entity defining a relation, we consider it as a big-step entity defining a function:

2. we refunctionalize [13] the simplified hygienic abstract machine of Item 1 into a continuation-passing evaluation function, which we write back to direct style, obtaining a functional program that is correct by construction and that implements a heapless natural semantics with the same built-in hygiene;
3. in addition, we observe that the evaluation function implementing this hygienic natural semantics is in defunctionalized form [14], and we present the corresponding higher-order evaluation function.

Overview. We start with a call-by-name semantics of the λ_{let}-calculus (Section 2). This reduction semantics provides a syntactic account of demand-driven computation. Extending this syntactic account with the memoization of intermediate results yields Ariola et al.'s call-by-need semantics of the λ_{let}-calculus (Section 3). This reduction semantics is deceivingly concise: in the first half of this article (Section 4), we methodically analyze it, considering in turn its potential redexes (Section 4.1), its (lack of) hygiene (Section 4.2), its evaluation contexts (Section 4.3), the recomposition of its evaluation contexts around a term (Section 4.4), its decomposition of a non-answer term into a potential redex and its evaluation context according to the reduction strategy (Section 4.5), its contraction rules (Section 4.6), its standard one-step reduction (Section 4.7), and its standard reduction-based evaluation (Section 4.8). The extensional properties such as unique decomposition, standardization, and hygiene ensure the existence of a deterministic evaluator extensionally. However, it is our thesis that they also provide precious intensional guidelines. We illustrate this thesis in the second half of this article (Sections 5 to 8): from the reduction semantics, we mechanically derive an abstract machine (Section 5), from this abstract machine, we mechanically derive a natural semantics (Sections 7.1 and 7.2), and from this natural semantics we mechanically derive a higher-order evaluation function (Section 7.3).

The ML code of the entire derivation is available from the last author's web page.[2]

Prerequisites. We assume a degree of familiarity with the formats of operational semantics – specifically reduction semantics, abstract machines, and natural semantics – though no more as can be gathered, e.g., in the first author's lecture notes at AFP 2008 [15].

2. The standard call-by-name reduction for the λ-calculus

Let us start with demand-driven computation and the standard reduction corresponding to call by name. The call-by-name reduction semantics for the λ_{let}-calculus reads as follows:

Definition 1 (call-by-name λ_{let}-calculus)**.**

Syntax:

$$
\begin{array}{rl}
\mathsf{Var} \ni & x \\
\mathsf{Term} \ni & T ::= x \mid \lambda x.T \mid T\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ T \\
\mathsf{Value} \ni & V ::= \lambda x.T \\
\mathsf{Answer} \ni & A ::= V \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A \\
\mathsf{Evaluation\ Context} \ni & E ::= [\,] \mid E\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E
\end{array}
$$

Contraction rules:

$$
\begin{array}{lrcl}
(I) & (\lambda x.T)\ T_1 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T \\
(N) & \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E[x] & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E[T] \\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2
\end{array}
$$

In words:

[2] `http://www.zerny.dk/def-int-for-call-by-need.html`

- Programs are closed λ-terms with no let expressions.
- Terms are pure λ-terms with non-recursive let expressions. (We follow the tradition of referring to λ-declared and let-declared denotables as "variables" even though they do not vary.)
- Values are λ-abstractions.
- Answers are let expressions nested around a value.
- Evaluation contexts are terms with a hole that are constructed inductively. The notation "$E[T]$" stands for a term that decomposes into an evaluation context E and a term T. Evaluation contexts specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-name reduction strategy.

Each contraction rule maps a redex to a contractum:

- Rule (I) introduces a let binding from an application, in a way akin to let insertion in partial evaluation [16].
- Rule (N) hygienically substitutes a definiens (here: a term) for the occurrence of a let-declared variable arising in an evaluation context. There may be more than one occurrence of the variable in the context. These other occurrences are not substituted.
- Rule (C) allows let bindings to commute with applications, hygienically, i.e., renaming what needs to be renamed so that no free variable is captured.

Reduction is then defined in terms of evaluation contexts and contraction. A term T_0 reduces to T_1 if there exists an evaluation context E, a redex T_0' and a contractum T_1' such that $T_0 = E[T_0']$, $(T_0', T_1') \in (I) \cup (N) \cup (C)$, and $T_1 = E[T_1']$. The following reduction sequence (one reduct per line) illustrates the demand-driven aspect of call by name as well as the duplication of work it entails. We note one-step reduction with $\mapsto_{\text{name}}$ and annotate each reduction step with the name of the corresponding contraction rule:

$$
\begin{array}{ll}
\underline{(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x))} \mapsto_{\text{name}} & (I) \\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ z}\ z \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ (\underline{(\lambda y.y)\ (\lambda x.x)})\ z \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ (\underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ y})\ z \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \underline{(\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)\ z} \mapsto_{\text{name}} & (C) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{(\lambda x.x)\ z} \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ x} \mapsto_{\text{name}} & (N) \\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ z} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \underline{(\lambda y.y)\ (\lambda x.x)} \mapsto_{\text{name}} & (I) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \underline{\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ y} \mapsto_{\text{name}} & (N) \\
\mathsf{let}\ z\ \mathsf{be}\ (\lambda y.y)\ (\lambda x.x)\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ z\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x &
\end{array}
$$

At every step, we have explicitly decomposed each reduct into a redex (underlined) and its evaluation context (not underlined). Each (N) contraction is triggered by a demand

over a variable: we have shaded the occurrence of this variable. Each of the two shaded occurrences of z forces the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is not memoized.

3. The standard call-by-need reduction for the λ-calculus

Let us supplement demand-driven computation with the memoization of intermediate results to obtain the standard reduction corresponding to call by need. The following call-by-need reduction semantics for the λ_{let}-calculus is common to Ariola, Felleisen, Maraist, Odersky, and Wadler's articles [1, 2, 17], renaming non-terminals for notational uniformity:

Definition 2 (call-by-need λ_{let}-calculus [17, Figure 3])**.**

Syntax:

$$
\begin{array}{rcl}
\mathsf{Var} \ni x & & \\
\mathsf{Term} \ni T & ::= & x \mid \lambda x.T \mid T\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ T \\
\mathsf{Value} \ni V & ::= & \lambda x.T \\
\mathsf{Answer} \ni A & ::= & V \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A \\
\mathsf{Evaluation\ Context} \ni E & ::= & [\,] \mid E\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E[x]
\end{array}
$$

Contraction rules:

$$
\begin{array}{lrcl}
(I) & (\lambda x.T)\ T_1 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T \\
(V) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2 \\
(A) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ \mathsf{in}\ E[x] & \rightarrow & \mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E[x]
\end{array}
$$

In words:

- Programs are closed λ-terms with no let expressions.
- Terms are pure λ-terms with non-recursive let expressions.
- Values are λ-abstractions.
- Answers are let expressions nested around a value.
- Evaluation contexts are terms with a hole that are constructed inductively. They specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-need reduction strategy The notation "$E[T]$" stands for a term that decomposes into an evaluation context E and a term T. Evaluation contexts specify where in a term the contraction rules can be applied. In the present case, the evaluation contexts specify the call-by-need reduction strategy.

Each contraction rule maps a redex to a contractum:

- Rule (I) introduces a let binding from an application.

- Rule (V) hygienically substitutes a definiens (here: a value) for the occurrence of a let-declared variable arising in an evaluation context. There may be more than one occurrence of the variable in the context. These other occurrences are not substituted.
- Rule (C) allows let bindings to commute with applications.
- Rule (A) re-associates let bindings.

Where call by name uses Rule (N), call by need uses Rule (V), ensuring that only values are duplicated. The reduction strategy thus also differs, so that the definiens of a needed variable is first reduced and this variable is henceforth declared to denote this reduct.

The following reduction sequence (one reduct per line) illustrates the demand-driven aspect of call by need as well as the memoization of intermediate results it enables. We note one-step reduction with $\mapsto_{\text{need}}$ (and specify it precisely in Section 4.7) and annotate each reduction step with the name of the corresponding contraction rule:

$$\begin{array}{ll}
\underline{(\lambda z.z\ z)\ ((\lambda y.y)\ (\lambda x.x))} \mapsto_{\text{need}} & (I)\\
\mathsf{let}\ z\ \mathsf{be}\ \underline{(\lambda y.y)\ (\lambda x.x)}\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (I)\\
\mathsf{let}\ z\ \mathsf{be}\ \underline{(\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{y})}\ \mathsf{in}\ \boxed{z}\ z \mapsto_{\text{need}} & (V)\\
\underline{\mathsf{let}\ z\ \mathsf{be}\ (\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x)\ \mathsf{in}\ \boxed{z}\ z} \mapsto_{\text{need}} & (A)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{z}\ z} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{(\lambda x.x)\ z} \mapsto_{\text{need}} & (I)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \boxed{z}\ \mathsf{in}\ \boxed{x}} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \underline{\mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \boxed{x}} \mapsto_{\text{need}} & (V)\\
\mathsf{let}\ y\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ z\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ \lambda x.x\ \mathsf{in}\ \lambda x.x &
\end{array}$$

At every step, we have explicitly decomposed each reduct into a redex (underlined) and its evaluation context (not underlined). We have shaded the occurrences of the variables whose value is needed in the course of the reduction. Only the first shaded occurrence of z forces the reduction of $(\lambda y.y)\ (\lambda x.x)$. The result of this demand-driven reduction is memoized in the let expression that declares z. It is thus reused when z triggers the two subsequent (V) contractions. This let expression is needed as long as z occurs free in its body; thereafter it can be elided with a garbage-collection rule [18].

This enumeration of successive call-by-need reducts is shorter than the call-by-name reduction sequence in Section 2: call by need is an optimization of call by name [1, 2].

—

To add computational intuition and also to make it easier to test our successive implementations, we take the liberty of extending the calculus of Definition 2 with integers and the (strict) successor function:

Definition 3 (call-by-need λ_{let}-calculus applied to integers)**.**

Syntax:

$$\begin{array}{rcl}
\mathsf{Term} \ni T & ::= & \ulcorner n \urcorner \mid \mathit{succ}\ T \mid x \mid \lambda x.T \mid T\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ T\\
\mathsf{Value} \ni V & ::= & \ulcorner n \urcorner \mid \lambda x.T\\
\mathsf{Answer} \ni A & ::= & V \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A\\
\mathsf{Evaluation\ Context} \ni E & ::= & [\,] \mid \mathit{succ}\ E \mid E\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E \mid \mathsf{let}\ x\ \mathsf{be}\ E\ \mathsf{in}\ E[x]
\end{array}$$

Contraction rules:

$$
\begin{array}{lrcl}
(I) & (\lambda x.T)\ T_1 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T \\
(I') & succ\ \ulcorner n \urcorner & \rightarrow & \ulcorner n' \urcorner \quad \text{where } n' = n + 1 \\
(V) & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[x] & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E[V] \\
(C) & (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2 & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2 \\
(C') & succ\ (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A) & \rightarrow & \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ succ\ A \\
(A) & \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1 & \rightarrow & \mathsf{let}\ y\ \mathsf{be}\ T_1 \\
 & \mathsf{in}\ A & & \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A \\
 & \mathsf{in}\ E[x] & & \mathsf{in}\ E[x]
\end{array}
$$

Compared to Definition 2, the shaded parts are new.

This definition is our starting point.

4. Some exegesis

Definition 3 packs a lot of information. Let us methodically spell it out:

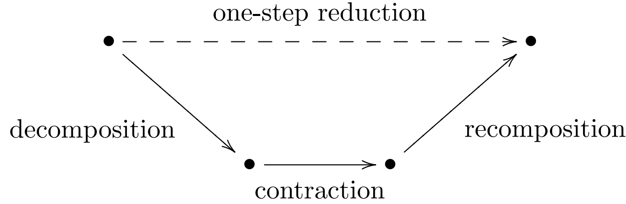
- The contraction rules are a mouthful, and so in Section 4.1, we identify their underlying structure by stating a grammar for potential redexes.
- In reduction semantics, evaluation is defined as iterated one-step reduction. However, one-step reduction assumes Barendregt's variable convention, i.e., that all declared variables are distinct, but not all the contraction rules preserve this convention: naive iteration is thus unsound. Rather than subsequently ensuring hygiene as in Garcia et al.'s construction of a lazy abstract machine [4], we make the contraction rules explicitly hygienic in Section 4.2 to make one-step reduction preserve Barendregt's variable convention upfront.
- The evaluation contexts are unusual in that they involve terms that are uniquely decomposable into a delimited evaluation context and a variable. In Section 4.3, we restate their definition to clearly distinguish between ordinary evaluation contexts and delimited evaluation contexts.
- The one-step reduction of a reduction semantics is implicitly structured in three parts: given a non-answer term,

  (1, decomposition): locate the next potential redex according to the reduction strategy;

  (2, contraction): if the potential redex is an actual one, i.e., if the non-answer term is not stuck, contract this actual redex as specified by the contraction rules; and

  (3, recomposition): fill the surrounding context with the contractum to construct the next term in the reduction sequence.

  Diagrammatically:



Based on Sections 4.1, 4.2, and 4.3, we specify decomposition, hygienic contraction, and recomposition in Sections 4.4, 4.5, and 4.6. We then formalize hygienic one-step reduction in Section 4.7 and hygienic evaluation as iterated one-step reduction in Section 4.8.

4.1. Potential redexes

To bring out the underlying structure of the contraction rules, let us state a grammar for potential redexes:

$$\textsf{Potential Redex} \ni R ::= \mathit{succ}\ A \ \mid\ A\ T \ \mid\ \textsf{let}\ x\ \textsf{be}\ A\ \textsf{in}\ E[x]$$

where $E[x]$ stands for a non-answer term.

The two forms of answers – value and let expression – give rise to one contraction rule for each production in the grammar of potential redexes:

- (I') arises from the application of the successor function to a value; (C') arises from the application of the successor function to a let expression; likewise,
- (I) and (C) arise from the application of an answer; and
- (V) and (A) arise from the binding of an answer to a variable whose value is needed.

Not all potential redexes are actual ones: a non-answer term may be stuck due to a type error.

4.2. Barendregt's variable convention

The definition of evaluation as iterated one-step reduction assumes Barendregt's variable convention, i.e., that all bound variables are distinct. Indeed the rules (V), (C) and (A) assume the variable convention when they move a term in the scope of a binding. A reduction step involving (V), however, yields a term where the variable convention does not hold, since V is duplicated and it may contain λ-abstractions and therefore bound variables.

There are many ways to ensure variable hygiene, if not the variable convention, at all times. We choose to allow λ-declared (not let-declared) variables to overlap, since no reduction can take place inside a λ-abstraction prior to its application, and to ensure that all let-declared variables are distinct. To this end, in Rule (I), we make each let expression explicitly hygienic by declaring a globally fresh variable and renaming the corresponding λ-declared variable in passing:

$$(I)\quad (\lambda x.T)\ T_1 \rightarrow \textsf{let}\ x'\ \textsf{be}\ T_1\ \textsf{in}\ T[x'/x] \quad \text{where } x' \text{ is fresh}$$

This explicit hygiene ensures Barendregt's variable convention for let-declared variables.

Other alternatives exist for ensuring variable hygiene. We have explored several of them, and in our experience they lead to semantic artifacts that are about as simple and understandable as the ones presented here. The alternative we chose here, i.e., making Rule (I) explicitly hygienic corresponds to, and is derived into the same renaming side condition as in Maraist, Odersky, and Wadler's natural semantics [2, Figure 11]. We also observe that this alternative is at the heart of the renaming mechanism in Garcia et al.'s lazy abstract machine [4, Section 4.5]. Across small-step semantics (the present work), abstract machines (Garcia et al.), and big-step semantics (Maraist et al.), there is therefore a genuine consensus about what befits hygienic reduction best in call by need. We have characterized this consensus in Rule (I) here.

With Rule (I) explicitly hygienic, every contraction and thus every reduction step requires at most one fresh variable. Every finite reduction sequence (say, of length n) therefore requires at most n fresh variables. In fact, this notion of fresh variables is coinductive since programs may diverge and thus reduction sequences may be infinite. We thus materialize the freshness condition by threading a stream of fresh variables throughout successive contractions:

$$X \in \mathsf{FreshVars} = \nu X.\mathsf{Var} \times X$$

This stream is used to implement Rule (I):

$$(I) \quad ((x',\, X),\, (\lambda x.T)\ T_1) \to (X,\, \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x])$$

In all the other rules, X is threaded passively. Threading such a stream matches implementational practice, where the so-called "gensym" procedure yields a fresh variable. Here, this fresh variable is the next one in the stream.

4.3. The evaluation contexts

The grammars of contexts for call by need, in Definitions 2 and 3, are unusual compared to the one for call by name given in Definition 1. Call-by-need evaluation contexts have an additional constructor involving the term "$E[x]$" for which there exists a variable x in the eye of a delimited context E. Spelling out decomposition (see Section 4.5 and Figure 3) shows that these delimited contexts are inductively constructed *outside in* whereas all the others are constructed *inside out*. To emphasize the computational difference we make it explicit which are which by adopting two isomorphic representations of contexts as a list of frames:

$$\begin{array}{rcl} \mathsf{Context\ Frame} \ni F & ::= & \mathit{succ}\ \Box \mid \Box\ T \mid \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \Box \mid \mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x] \\ \mathsf{Outside\text{-}in\ Context} \ni E^{oi} & ::= & \varepsilon^{oi} \mid E^{oi} :: F \\ \mathsf{Inside\text{-}out\ Context} \ni E^{io} & ::= & \varepsilon^{io} \mid F :: E^{io} \end{array}$$

Here $\Box$ is the hole in a context frame, ε^{oi} is the empty outside-in context, ε^{io} is the empty inside-out context, and $::$ is the (overloaded) context constructor. For example, the context $E = ([\,]\ T_1)\ T_2$ is equivalent to the outside-in context $E^{oi} = \varepsilon^{oi} :: (\Box\ T_1) :: (\Box\ T_2)$ and to the inside-out context $E^{io} = (\Box\ T_1) :: (\Box\ T_2) :: \varepsilon^{io}$ in the sense that for a term T_0 they all recompose to $(T_0\ T_1)\ T_2$, as defined in Section 4.4. Outside-in contexts hang to

$$\frac{}{\langle T,\, \varepsilon^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T} \qquad \frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_1}{\langle T,\, E^{oi} :: (\mathit{succ}\ \Box)\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathit{succ}\ T_1} \qquad \frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_0}{\langle T,\, E^{oi} :: (\Box\ T_1)\rangle_{oi} \Uparrow_{\mathrm{rec}} T_0\ T_1}$$

$$\frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, E^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box)\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

$$\frac{\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_1 \qquad \langle x,\, E_1^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, E^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E_1^{oi}[x])\rangle_{oi} \Uparrow_{\mathrm{rec}} \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T_2}$$

Figure 1: Recomposition of outside-in contexts

$$\frac{}{\langle T,\, \varepsilon^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T} \qquad \frac{\langle \mathit{succ}\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_1}{\langle T,\, (\mathit{succ}\ \Box) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_1} \qquad \frac{\langle T_0\ T_1,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T_0,\, (\Box\ T_1) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

$$\frac{\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

$$\frac{\langle x,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T \qquad \langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}{\langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T_2}$$

Figure 2: Recomposition of inside-out contexts

the left and inside-out contexts hang to the right. They are composed by concatenation to the left or to the right:

$$\begin{aligned} E^{oi} \circ_{oi} \varepsilon^{io} &= E^{oi} & \varepsilon^{oi} \circ_{io} E^{io} &= E^{io} \\ E^{oi} \circ_{oi} (F :: E^{io}) &= (E^{oi} :: F) \circ_{oi} E^{io} & (E^{oi} :: F) \circ_{io} E^{io} &= E^{oi} \circ_{io} (F :: E^{io}) \end{aligned}$$

NB. In this BNF of context frames, as pointed out in Section 2 and 3, the notation "$E^{oi}[x]$" represents a term that uniquely decomposes into an outside-in evaluation context E^{oi} and a variable x. In Section 5.2 and onwards, we take notational advantage of this paired representation to short-cut any subsequent decomposition of this term into E^{oi} and x.

4.4. Recomposition

Outside-in contexts and inside-out contexts are recomposed (or again are 'plugged' or 'filled') as follows:

Definition 4 (recomposition of outside-in contexts)**.** An outside-in context E^{oi} is recomposed around a term T into a term T' whenever $\langle T,\, E^{oi}\rangle_{oi} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 1.)

Definition 5 (recomposition of inside-out contexts)**.** An inside-out context E^{io} is recomposed around a term T into a term T' whenever $\langle T,\, E^{io}\rangle_{io} \Uparrow_{\mathrm{rec}} T'$ holds. (See Figure 2.)

$$
\begin{array}{rcl}
\langle \ulcorner n \urcorner,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle E^{io},\, \ulcorner n \urcorner \rangle_{context} \\
\langle succ\ T,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle T,\, (succ\ \Box) :: E^{io} \rangle_{term} \\
\langle x,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle E^{io},\, (\varepsilon^{oi},\, x) \rangle_{reroot} \\
\langle \lambda x.T,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle E^{io},\, \lambda x.T \rangle_{context} \\
\langle T_0\ T_1,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle T_0,\, (\Box\ T_1) :: E^{io} \rangle_{term} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io} \rangle_{term} & \downarrow_{\text{dec}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io} \rangle_{term} \\
\\
\langle \varepsilon^{io},\, A \rangle_{context} & \downarrow_{\text{dec}} & \langle A \rangle_{answer} \\
\langle (succ\ \Box) :: E^{io},\, A \rangle_{context} & \downarrow_{\text{dec}} & \langle succ\ A,\, E^{io} \rangle_{redex} \\
\langle (\Box\ T_1) :: E^{io},\, A \rangle_{context} & \downarrow_{\text{dec}} & \langle A\ T_1,\, E^{io} \rangle_{redex} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io},\, A \rangle_{context} & \downarrow_{\text{dec}} & \langle E^{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io},\, A \rangle_{context} & \downarrow_{\text{dec}} & \langle \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E^{oi}[x],\, E^{io} \rangle_{redex} \\
\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box) :: E^{io},\, (E^{oi},\, x) \rangle_{reroot} & \downarrow_{\text{dec}} & \langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]) :: E^{io} \rangle_{term} \\
\langle F :: E^{io},\, (E^{oi},\, x) \rangle_{reroot} & \downarrow_{\text{dec}} & \langle E^{io},\, (E^{oi} :: F,\, x) \rangle_{reroot} \\
 & & \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \Box
\end{array}
$$

Figure 3: Decomposition of an answer term into itself
and of a non-answer term into a potential redex and its evaluation context

For example, let us recompose the term $\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2$ in the inside-out context $(\Box\ T_3) :: \varepsilon^{io}$:

$$
\begin{array}{c}
\langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2,\, (\Box\ T_3) :: \varepsilon^{io} \rangle_{io} \\
\text{recomposition} \Big\downarrow \Uparrow_{\text{rec}} \\
(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2)\ T_3
\end{array}
$$

Proposition 6 (unique recomposition of outside-in contexts)**.** For any term T and outside-in context E^{oi} such that $\langle T,\, E^{oi} \rangle_{oi} \Uparrow_{\text{rec}} T'$ holds, the term T' is unique.

Proof. Induction on E^{oi}. □

Proposition 7 (unique recomposition of inside-out contexts)**.** For any term T and inside-out context E^{io} such that $\langle T,\, E^{io} \rangle_{io} \Uparrow_{\text{rec}} T'$ holds, the term T' is unique.

Proof. Induction on E^{io}. □

4.5. Decomposition

Decomposing a non-answer term into a potential redex and its evaluation context according to the reduction strategy is at the heart of a reduction semantics, but outside of the authors' publications, it seems never to be spelled out. Let us do so.

There are many ways to specify decomposition. In our experience, a convenient one is the abstract machine displayed in Figure 3. This machine starts in the configuration $\langle T,\, \varepsilon^{io} \rangle_{term}$, for a given term T. It halts in an answer state if the given term contains no potential redex, and in a decomposition state $\langle R,\, E^{io} \rangle_{redex}$ otherwise, where R is a potential redex in T and E^{io} its evaluation context according to the reduction strategy specified by the grammar of evaluation contexts.

Definition 8 (decomposition)**.** The decomposition relation, $\downarrow^*_{\mathrm{dec}}$, is the transitive closure of $\downarrow_{\mathrm{dec}}$. (See Figure 3.)

For example, let us decompose the non-answer term (let x be $\lambda x_0.x_0$ in $(x\ T_1)\ T_2)\ T_3$:

$$\begin{array}{c} \langle(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3,\ \varepsilon^{io}\rangle_{term} \\ \text{decomposition} \Big\downarrow \downarrow^*_{\mathrm{dec}} \\ \langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\square\ T_1) :: (\square\ T_2))[x],\ (\square\ T_3) :: \varepsilon^{io}\rangle_{redex} \end{array}$$

The *term* and *context* transitions are traditional: one dispatches on a term and the other on the top context frame. The *reroot* transitions locate the let-binder for a variable while maintaining the outside-in context from the binder to its occurrence, zipper-style [19].[3] In effect, the transitions reverse the prefix of an inside-out context into an outside-in context. For the example above, this reversal is carried out in the following sub-steps of decomposition:

$$\begin{array}{c} \langle(\square\ T_1) :: (\square\ T_2) :: (\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ \square) :: (\square\ T_3) :: \varepsilon^{io},\ (\varepsilon^{io},\ x)\rangle_{reroot} \\ \Big\downarrow \downarrow^*_{\mathrm{dec}} \\ \langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\square\ T_1) :: (\square\ T_2))[x],\ (\square\ T_3) :: \varepsilon^{io}\rangle_{redex} \end{array}$$

Proposition 9 (vacuous decomposition of an answer term)**.** An answer term is vacuously decomposed into itself: for any answer term A, $\langle A,\ \varepsilon^{io}\rangle_{term} \downarrow^*_{\mathrm{dec}} \langle A\rangle_{answer}$ holds.

Proof. By induction: the transitions over *term*-configurations turn the answer term inside-out into a context until its innermost value is reached, and the transitions over *context*-configurations turn back this context inside-out into the answer term until the empty context is reached. ☐

Proposition 10 (unique decomposition of a non-answer term)**.** For any non-answer term T such that $\langle T,\ \varepsilon^{io}\rangle_{term} \downarrow^*_{\mathrm{dec}} \langle R,\ E^{io}\rangle_{redex}$ holds, the potential redex R and evaluation context E^{io} are unique.

Proof. The $\downarrow_{\mathrm{dec}}$ relation is uniquely determined and $\langle R,\ E^{io}\rangle_{redex}$ is a terminal state, thus by transitivity $\langle R,\ E^{io}\rangle_{redex}$ is unique. ☐

4.6. The contraction rules

In accordance with the new BNF of contexts, the contraction rules of Definition 3 are hygienically stated as follows:

[3] Decomposition could be stuck for terms containing free variables, but we assume programs to be closed.

$$
\begin{array}{llll}
(I) & ((x',\, X),\, (\lambda x.T)\; T_1) \to (X,\, \mathsf{let}\; x' \;\mathsf{be}\; T_1 \;\mathsf{in}\; T[x'/x]) & \\
(I') & (X,\, succ\, \ulcorner n \urcorner) \to (X,\, \ulcorner n' \urcorner) & \text{where } n' = n + 1 \\
(V) & (X,\, \mathsf{let}\; x \;\mathsf{be}\; V \;\mathsf{in}\; E^{oi}[x]) \to (X,\, \mathsf{let}\; x \;\mathsf{be}\; V \;\mathsf{in}\; T) & \text{where } \langle V,\, E^{oi} \rangle_{oi} \Uparrow_{\mathrm{rec}} T \\
(C) & (X,\, (\mathsf{let}\; x \;\mathsf{be}\; T_1 \;\mathsf{in}\; A)\; T_2) \to (X,\, \mathsf{let}\; x \;\mathsf{be}\; T_1 \;\mathsf{in}\; A\; T_2) & \\
(C') & (X,\, succ\; (\mathsf{let}\; x \;\mathsf{be}\; T \;\mathsf{in}\; A)) \to (X,\, \mathsf{let}\; x \;\mathsf{be}\; T \;\mathsf{in}\; succ\; A) & \\
(A) & (X,\, \mathsf{let}\; x \;\mathsf{be}\; \mathsf{let}\; y \;\mathsf{be}\; T_1 \;\mathsf{in}\; A \;\mathsf{in}\; E^{oi}[x]) \to (X,\, \mathsf{let}\; y \;\mathsf{be}\; T_1 \;\mathsf{in}\; \mathsf{let}\; x \;\mathsf{be}\; A \;\mathsf{in}\; E^{oi}[x]) &
\end{array}
$$

Definition 11 (notion of reduction)**.** $\mathcal{R} = (I) \cup (I') \cup (V) \cup (C) \cup (C') \cup (A)$ and a redex R contracts to T, denoted $R \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} T$, iff $((X,\, R),\, (X',\, T)) \in \mathcal{R}$.

For example, let us contract the actual redex $\mathsf{let}\; x \;\mathsf{be}\; \lambda x_0.x_0 \;\mathsf{in}\; (\varepsilon^{oi} :: (\square\; T_1) :: (\square\; T_2))[x]$ with the stream of fresh variables X:

$$
\begin{array}{c}
\mathsf{let}\; x \;\mathsf{be}\; \lambda x_0.x_0 \;\mathsf{in}\; (\varepsilon^{oi} :: (\square\; T_1) :: (\square\; T_2))[x] \\
\text{contraction of } (V) \Big\downarrow \overset{X;X}{\rightsquigarrow}_{\mathcal{R}} \\
\mathsf{let}\; x \;\mathsf{be}\; \lambda x_0.x_0 \;\mathsf{in}\; ((\lambda x_0.x_0)\; T_1)\; T_2
\end{array}
$$

Proposition 12 (unique contraction)**.** For any redex R and stream of fresh variables X such that $R \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} T$ holds, T and X' are unique.

Proof. By case analysis on R. (See Section 4.1.) □

4.7. Standard one-step reduction

The standard one-step reduction performs one contraction in a non-answer term and is defined as

1. locating a potential redex and its evaluation context in the non-answer term through a number of decomposition steps,
2. contracting this potential redex if it is an actual one (otherwise the non-answer term is stuck), and
3. recomposing the resulting contractum into the evaluation context:

Definition 13 (standard one-step reduction)**.**

$$
(X,\, T) \mapsto_{\mathrm{need}} (X',\, T'') \text{ iff } \begin{cases} \langle T,\, \varepsilon^{oi} \rangle_{term} \downarrow^{*}_{\mathrm{dec}} \langle R,\, E^{io} \rangle_{redex} \\ R \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} T' \\ \langle T',\, E^{io} \rangle_{io} \Uparrow_{\mathrm{rec}} T'' \end{cases}
$$

Note that the standard one-step reduction is not the compatible closure of $\mathcal{R}$. The compatible closure, $\to_{\mathcal{R}}$, is closed over general contexts (i.e., terms with a hole), whereas the standard one-step reduction is closed over the restricted grammar of evaluation contexts.

For example, given a stream of fresh variables X, let us illustrate standard one-step reduction for the term $(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3$:

$$\begin{array}{c}
\langle(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (x\ T_1)\ T_2)\ T_3,\ \varepsilon^{io}\rangle_{term} \\
\text{decomposition} \big\downarrow \downarrow^*_{\text{dec}} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ (\varepsilon^{oi} :: (\square\ T_1) :: (\square\ T_2))[x],\ (\square\ T_3) :: \varepsilon^{io}\rangle_{redex} \\
\text{contraction of } (V) \big\downarrow \langle \overset{X;X}{\rightsquigarrow}_{\mathcal{R}},\ id\rangle \\
\langle \mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2,\ (\square\ T_3) :: \varepsilon^{io}\rangle_{io} \\
\text{recomposition} \big\downarrow \Uparrow_{\text{rec}} \\
(\mathsf{let}\ x\ \mathsf{be}\ \lambda x_0.x_0\ \mathsf{in}\ ((\lambda x_0.x_0)\ T_1)\ T_2)\ T_3
\end{array}$$

Proposition 14 (unique standard one-step reduction)**.** For any term T and stream of fresh variables X such that $(X, T) \mapsto_{\text{need}} (X', T')$ holds, T' and X' are unique.

Proof. Corollary of unique decomposition (Proposition 10), unique contraction (Proposition 12), and unique recomposition (Proposition 7). □

4.8. Standard reduction-based evaluation

The standard reduction-based evaluation is defined as the iteration of the standard one-step reduction. It thus enumerates the successive call-by-need reducts, i.e., the standard reduction sequence, of any given term:

Definition 15 (standard reduction-based evaluation)**.** Standard reduction-based evaluation, $\mapsto^*_{\text{need}}$, is the reflexive, transitive closure of standard one-step reduction, $\mapsto_{\text{need}}$.

Proposition 16 (unique standard reduction-based evaluation to answers)**.** For any term T and stream of fresh variables X such that $(X, T) \mapsto^*_{\text{need}} (X', A)$ holds, A and X' are unique.

Proof. Corollary of unique standard reduction (Proposition 14). □

4.9. Conclusion and perspectives

As illustrated here, there is substantially more than meets the eye in a reduction semantics.

In addition, extensional properties such as unique decomposition, standardization, and hygiene do not only ensure the existence of a deterministic evaluator extensionally, but it is our thesis that they also provide precious intensional guidelines. Indeed, after exegetically spelling out what does not readily meet the eye, things become compellingly simple:

- refocusing the standard reduction-based evaluation immediately gives a reduction-free abstract machine (Section 5.1) and compressing the corridor transitions of this abstract machine improves the efficiency of its execution (Section 5.2);

- we can then move from the relational view of small-step abstract machines to the functional view of big-step abstract machines (Section 6);
- refunctionalizing the compressed big-step abstract machine with respect to the evaluation contexts gives a reduction-free evaluation function in continuation-passing style (Section 7.1). Mapping this evaluation function back to direct style gives a functional implementation of a natural semantics (Section 7.2).[4]

All of these semantic artifacts are correct by construction, and their operational behaviors rigorously mirror each other in a lock-step sort of way. For one example, the semantic artifacts agree not only up to α-equivalence but up to syntactic equality. For another example, should one be tempted to fine-tune either of these semantic artifacts, one is then in position to adjust the others to keep their operational behaviors in line, or to understand why this alignment is not possible and where coherence got lost in the fine-tuning [15].

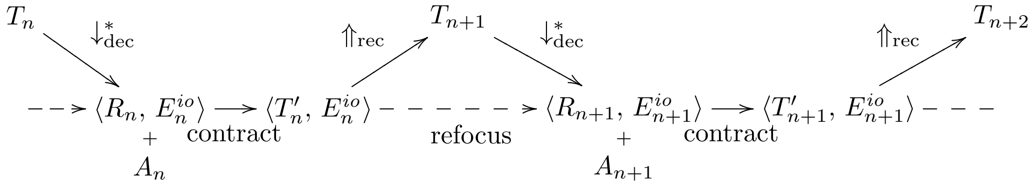
5. From reduction semantics to abstract machine

This section implements the first half of the programme outlined in Section 4.9. We first go from the standard reduction-based evaluation of Definition 15 (that enumerates all the successive reducts in the standard reduction sequence) to a reduction-free evaluation (that does not perform this enumeration because all the reducts are deforested away). This reduction-free evaluation takes the form of an abstract machine.

5.1. Refocusing: from reduction semantics to abstract machine

By recomposing and then immediately decomposing, a reduction-based evaluator takes a detour from a redex site, up to the top of the term, and back down again to the next redex site. The steps that make up this detour can be eliminated by refocusing [12]. Refocusing the reduction-based evaluation of a reduction semantics yields a reduction-free evaluation that directly navigates in a term from redex site to redex site without any detour via the top of the term.

Refocusing replaces successive recompositions and decompositions by a 'refocus' relation that associates a contractum and its (inside-out) evaluation context to an answer or a decomposition consisting of the next potential redex and associated evaluation context:



[4] Recently [20], Pirog and Biernacki have used the CPS transformation and defunctionalization to connect Launchbury and Sestoft's natural semantics for lazy evaluation [5, 7] and Peyton Jones's spineless tagless G-machine [21].

Figure 4 displays the naive, reduction-based definition of refocusing: an evaluation context is recomposed around a contractum and the resulting reduct is decomposed either into an answer or into another potential redex and its evaluation context. This definition is 'reduction-based' because the intermediate reduct is constructed.

$$\langle T,\, E^{io}\rangle_{term} \rightarrow_{\text{refocus}} D \text{ iff } \langle T,\, E^{io}\rangle_{io} \Uparrow_{\text{rec}} T' \;\wedge\; \langle T',\, \varepsilon^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$$

Figure 4: Reduction-based refocusing

Surprisingly, *optimal refocusing consists of simply continuing with decomposition from the contractum and its associated evaluation context, according to the standard reduction strategy* [12], here as well as in all the reduction semantics in Felleisen and Flatt's lecture notes on programming languages and lambda calculi [22]. (This is another reason why we place such store in the decomposition function of a reduction semantics.)

Figure 5 displays the optimal, reduction-free definition of refocusing: a contractum and an evaluation context are directly associated with an answer or another potential redex and its evaluation context simply by decomposing the contractum in the evaluation context. This definition is 'reduction-free' because no intermediate reduct is constructed.

$$\langle T,\, E^{io}\rangle_{term} \rightarrow_{\text{refocus}} D \text{ iff } \langle T,\, E^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$$

Figure 5: Reduction-free refocusing

Reduction-free evaluation is defined, after an initial decomposition of the input term, as the iteration of contraction and reduction-free refocusing (i.e., term decomposition in the current evaluation context):

Definition 17 (standard reduction-free evaluation)**.** Let $\rightarrow_{\text{step}}$ be one-step contraction and refocusing: $\xrightarrow{X;X'}_{\text{step}} = \downarrow^{*}_{\text{dec}} \cup \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} \downarrow^{*}_{\text{dec}}$, where $\langle R,\, E^{io}\rangle_{redex} \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} \downarrow^{*}_{\text{dec}} D$ iff $R \overset{X;X'}{\rightsquigarrow}_{\mathcal{R}} T \wedge \langle T,\, E^{io}\rangle_{term} \downarrow^{*}_{\text{dec}} D$. Standard reduction-free evaluation, $\rightarrow^{*}_{\text{step}}$, is the transitive closure of $\rightarrow_{\text{step}}$. Notationally we use $\xrightarrow{X;X'}{}^{*}_{\text{step}}$ to express that X is the input stream and X' is a suffix of X obtained after iterating $\rightarrow_{\text{step}}$.

Evaluation is thus defined with the decomposition transitions from Figure 3 plus, for each contraction rule from Section 4.6, one transition towards decomposing the contractum in the current evaluation context. Like decomposition in Figure 3, evaluation therefore takes the form of an abstract machine.[5] This abstract machine is displayed in Figure 6.

Proposition 18 (full correctness)**.** For the abstract machine of Figure 6,

$$(X,\, T) \mapsto^{*}_{\text{need}} (X',\, A) \;\Leftrightarrow\; \langle T,\, \varepsilon^{io}\rangle_{term} \xrightarrow{X;X'}{}^{*}_{\text{step}} \langle A\rangle_{answer}.$$

Proof. Corollary of the correctness of refocusing [12, 23]. □

[5] Giving decomposition another format would make evaluation inherit this format.

$$
\begin{array}{rcl}
\langle \ulcorner n \urcorner,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle E^{io},\, \ulcorner n \urcorner \rangle_{\mathit{context}} \\
\langle \mathit{succ}\ T,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle T,\, (\mathit{succ}\ \square) :: E^{io} \rangle_{\mathit{term}} \\
\langle x,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle E^{io},\, (\varepsilon^{oi},\, x) \rangle_{\mathit{reroot}} \\
\langle \lambda x.T,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle E^{io},\, \lambda x.T \rangle_{\mathit{context}} \\
\langle T_0\ T_1,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle T_0,\, (\square\ T_1) :: E^{io} \rangle_{\mathit{term}} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io} \rangle_{\mathit{term}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{\mathit{term}} \\
\\
\langle \varepsilon^{io},\, A \rangle_{\mathit{context}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle A \rangle_{\mathit{answer}} \\
\langle (\mathit{succ}\ \square) :: E^{io},\, A \rangle_{\mathit{context}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathit{succ}\ A,\, E^{io} \rangle_{\mathit{redex}} \\
\langle (\square\ T_1) :: E^{io},\, A \rangle_{\mathit{context}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle A\ T_1,\, E^{io} \rangle_{\mathit{redex}} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\, A \rangle_{\mathit{context}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle E^{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{\mathit{context}} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io},\, A \rangle_{\mathit{context}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ E^{oi}[x],\, E^{io} \rangle_{\mathit{redex}} \\
\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\, (E^{oi},\, x) \rangle_{\mathit{reroot}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io} \rangle_{\mathit{term}} \\
\langle F :: E^{io},\, (E^{oi},\, x) \rangle_{\mathit{reroot}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle E^{io},\, (E^{oi} :: F,\, x) \rangle_{\mathit{reroot}} \\
 & & \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square \\
\\
\langle \mathit{succ}\ \ulcorner n \urcorner,\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \ulcorner n' \urcorner,\, E^{io} \rangle_{\mathit{term}} \\
 & & \text{where } n' = n + 1 \\
\langle \mathit{succ}\ (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A),\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \mathit{succ}\ A,\, E^{io} \rangle_{\mathit{term}} \\
\langle (\lambda x.T)\ T_1,\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{(x',\, X);X}_{\mathrm{step}} & \langle \mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ T[x'/x],\, E^{io} \rangle_{\mathit{term}} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2,\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2,\, E^{io} \rangle_{\mathit{term}} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ E^{oi}[x],\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ T,\, E^{io} \rangle_{\mathit{term}} \\
 & & \text{where } \langle V,\, E^{oi} \rangle_{oi} \Uparrow_{\mathrm{rec}} T \\
\langle \mathsf{let}\ x\ \mathsf{be}\ \mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ \mathsf{in}\ E^{oi}[x],\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ A\ \mathsf{in}\ T,\, E^{io} \rangle_{\mathit{term}} \\
 & & \text{where } \langle x,\, E^{oi} \rangle_{oi} \Uparrow_{\mathrm{rec}} T
\end{array}
$$

Figure 6: Storeless abstract machine for call-by-need evaluation

5.2. *Transition compression: from abstract machine to abstract machine*

In the abstract machine of Figure 6, some of the transitions yield a configuration for which there unconditionally exists another transition: all transitions to a *term*-configuration with a known term, all transitions to a *context*-configuration with a known context, and all transitions to a *redex*-configuration with a known redex (i.e., all transitions to *redex*). For example, the application of any let expression, which is a redex, gives rise to the following unconditional transitions:

$$
\begin{array}{rcl}
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)\ T_2,\, E^{io} \rangle_{\mathit{redex}} & \xrightarrow{X;X}_{\mathrm{step}} & \langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A\ T_2,\, E^{io} \rangle_{\mathit{term}} \\
 & \xrightarrow{X;X}_{\mathrm{step}} & \langle A\ T_2,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{\mathit{term}} \\
 & \xrightarrow{X;X}_{\mathrm{step}} & \langle A,\, (\square\ T_2) :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{\mathit{term}}
\end{array}
$$

These so-called "corridor transitions" from one configuration to another can be hereditarily compressed so that the first configuration yields the last one in one transition.

Other transition compressions are determined by the structure of the term or of the evaluation context, and proceed over several steps. For example, analogously to what happens in Proposition 9, a *term*-configuration with an answer in a context always yields a *context*-configuration with this context and this answer:

$$\begin{array}{l}
\langle \mathsf{let}\ x_1\ \mathsf{be}\ T_1\ \mathsf{in}\ \mathsf{let}\ x_2\ \mathsf{be}\ T_2\ \mathsf{in}\ \cdots \mathsf{let}\ x_n\ \mathsf{be}\ T_n\ \mathsf{in}\ V,\ E^{io}\rangle_{term} \\
\xrightarrow{X;X}_{\text{step}} \langle \mathsf{let}\ x_2\ \mathsf{be}\ T_2\ \mathsf{in}\ \cdots \mathsf{let}\ x_n\ \mathsf{be}\ T_n\ \mathsf{in}\ V,\ (\mathsf{let}\ x_1\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io}\rangle_{term} \\
\ldots \\
\xrightarrow{X;X}_{\text{step}} \langle V,\ (\mathsf{let}\ x_n\ \mathsf{be}\ T_n\ \mathsf{in}\ \square) :: \cdots :: (\mathsf{let}\ x_2\ \mathsf{be}\ T_2\ \mathsf{in}\ \square) :: (\mathsf{let}\ x_1\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io}\rangle_{term} \\
\xrightarrow{X;X}_{\text{step}} \langle (\mathsf{let}\ x_n\ \mathsf{be}\ T_n\ \mathsf{in}\ \square) :: \cdots :: (\mathsf{let}\ x_2\ \mathsf{be}\ T_2\ \mathsf{in}\ \square) :: (\mathsf{let}\ x_1\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\ V\rangle_{context} \\
\ldots \\
\xrightarrow{X;X}_{\text{step}} \langle E^{io},\ \mathsf{let}\ x_1\ \mathsf{be}\ T_1\ \mathsf{in}\ \mathsf{let}\ x_2\ \mathsf{be}\ T_2\ \mathsf{in}\ \cdots \mathsf{let}\ x_n\ \mathsf{be}\ T_n\ \mathsf{in}\ V\rangle_{context}
\end{array}$$

So, rather than turning the answer inside-out into the prefix of a context (with transitions over *term*-configurations) until its innermost value is reached, and then turning this prefix inside-out back into the answer (with transitions over *context*-configurations), we can directly refocus the original *term*-configuration into the final *context*-configuration:

Proposition 19 (refocusing over answers)**.** For any A, E^{io} and X,

$$\langle A,\ E^{io}\rangle_{term} \xrightarrow{X;X\,*}_{\text{step}} \langle E^{io},\ A\rangle_{context}$$

Proof. Induction on A. □

Likewise, we can compress the transitions from a *term*-configuration over any term $E^{oi}[x]$ to a *term*-configuration over x, using the reverse concatenation "$\diamond$" defined in Section 4.3:

Proposition 20 (restoring outside-in evaluation contexts)**.** For any T, E^{io}, E^{oi} and X such that $\langle T,\ E^{oi}\rangle_{oi} \Uparrow_{\text{rec}} T'$,

$$\langle T',\ E^{io}\rangle_{term} \xrightarrow{X;X\,*}_{\text{step}} \langle T,\ E^{oi} \circ_{io} E^{io}\rangle_{term}$$

Proof. Induction on E^{oi}. □

Finally, we can short-cut the search for the definiens of a needed variable:

Proposition 21 (locating a definiens)**.** For any x, T, E^{oi}, E_1^{io}, E_2^{io}, and X, where x is not declared in E_1^{io},

$$\begin{array}{c}
\langle E_1^{io} :: (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ E^{oi}[x]) :: E_2^{io},\ (E^{oi},\ x)\rangle_{reroot} \\
\xrightarrow{X;X\,*}_{\text{step}} \\
\langle T,\ (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ (E^{oi} \circ_{oi} E_1^{io})[x]) :: E_2^{io}\rangle_{term}
\end{array}$$

Proof. Induction on E_1^{io}. □

The resulting abstract machine is displayed in Figure 7. No occurrences of "$\diamond$" appear in the final abstract machine because in the course of compression all occurrences introduced by Proposition 20 are subsequently eliminated by Proposition 21.

$$
\begin{array}{rcl}
\langle \ulcorner n \urcorner,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, \ulcorner n \urcorner \rangle_{context} \\
\langle succ\ T,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle T,\, (succ\ \square) :: E^{io} \rangle_{term} \\
\langle x,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, (\varepsilon^{oi},\, x) \rangle_{reroot} \\
\langle \lambda x.T,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, \lambda x.T \rangle_{context} \\
\langle T_0\ T_1,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle T_0,\, (\square\ T_1) :: E^{io} \rangle_{term} \\
\langle \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, E^{io} \rangle_{term} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{term} \\
\\
\langle \varepsilon^{io},\, A \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle A \rangle_{answer} \\
\langle (succ\ \square) :: E^{io},\, \ulcorner n \urcorner \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, \ulcorner n' \urcorner \rangle_{context} \\
 & & \text{where } n' = n + 1 \\
\langle (succ\ \square) :: E^{io},\, \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle (succ\ \square) :: (\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square) :: E^{io},\, A \rangle_{context} \\
\langle (\square\ T_1) :: E^{io},\, \lambda x.T \rangle_{context} & \overset{(x',\, X);X}{\longrightarrow}_{\mathrm{step}} & \langle T[x'/x],\, (\mathsf{let}\ x'\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io} \rangle_{term} \\
\langle (\square\ T_2) :: E^{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle (\square\ T_2) :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\, A \rangle_{context} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\, A \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, \mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A \rangle_{context} \\
\langle (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io},\, V \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle T,\, (\mathsf{let}\ x\ \mathsf{be}\ V\ \mathsf{in}\ \square) :: E^{io} \rangle_{term} \\
 & & \text{where } \langle V,\, E^{oi} \rangle_{oi} \Uparrow_{\mathrm{rec}} T \\
\langle \begin{pmatrix} \mathsf{let}\ x\ \mathsf{be}\ \square \\ \mathsf{in}\ E^{oi}[x] \end{pmatrix} :: E^{io},\, \begin{pmatrix} \mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \mathsf{in}\ A \end{pmatrix} \rangle_{context} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle \begin{pmatrix} \mathsf{let}\ x\ \mathsf{be}\ \square \\ \mathsf{in}\ E^{oi}[x] \end{pmatrix} :: \begin{pmatrix} \mathsf{let}\ y\ \mathsf{be}\ T_1 \\ \mathsf{in}\ \square \end{pmatrix} :: E^{io},\, A \rangle_{context} \\
\\
\langle (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \square) :: E^{io},\, (E^{oi},\, x) \rangle_{reroot} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle T_1,\, (\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]) :: E^{io} \rangle_{term} \\
\langle F :: E^{io},\, (E^{oi},\, x) \rangle_{reroot} & \overset{X;X}{\longrightarrow}_{\mathrm{step}} & \langle E^{io},\, (E^{oi} :: F,\, x) \rangle_{reroot} \\
 & & \text{where } F \neq \mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square
\end{array}
$$

Figure 7: The storeless abstract machine of Figure 6 after transition compression

Proposition 22 (full correctness)**.** For the abstract machine of Figure 7,

$$(X,\, T) \mapsto^{*}_{\mathrm{need}} (X',\, A) \quad \Leftrightarrow \quad \langle T,\, \varepsilon^{io} \rangle_{term} \overset{X;X'\,*}{\longrightarrow}_{\mathrm{step}} \langle A \rangle_{answer}.$$

Proof. By Proposition 18 and calculation using Propositions 19, 20, and 21. □

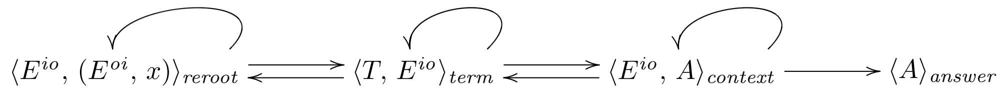
6. Small-step abstract machines define relations, big-step abstract machines define functions

A deterministic small-step abstract machine is characterized by a single-step state-transition system that associates a machine configuration with the next and is iterated toward a final state, if there is one. This characterization is aptly formalized by a *relation* that associates any non-final state to its successive states. The transitive closure of this relation then describes the transition sequence of any given term as well as its final state, if there is one. In contrast, a big-step abstract machine is characterized by a collection of mutually tail-recursive transitions mapping a configuration to a final state, if there is one. This characterization is aptly formalized by a *function* that maps any non-final

state to a final state, if there is one. Here we have no interest in the actual reduction sequences towards a final state.

The difference between the two styles of abstract machines is not typically apparent in the abstract-machine specifications found in programming-language semantics. A machine specification is normally presented as a small-step abstract machine given by reading the transition arrow as the definition of a single-step transition relation to be iterated and with the configuration labels as passive components of the configurations. However, the same specification can equally be seen as a big-step abstract machine if the transition labels are interpreted as tail-recursive functions, with the transition arrow connecting left- and right-hand sides of their definitions.

The following diagram depicts the states and transitions of the abstract machine in Figure 7:



States can be viewed as a sum type of labeled components, and the transition arrows as a relation that maps any non-final state to its successive states. Alternatively, the states can be viewed as a set of mutually (tail-)recursive functions and the transition arrows as tail calls between the functions. By Proposition 16 we know that final states are unique, and thus we can model the big-step abstract machine as a partial function mapping any term to its unique final state, if there is one.

These two views (of small steps and of big steps) are relevant to transform an abstract machine implementing an operational semantics into an interpreter implementing a natural semantics. Such interpreters operate in big steps, and it is for this reason that we now shift gears and view the abstract machine of Figure 7 as a big-step one with evaluation defined by a partial function. These two views on an abstract machine are illustrated in Appendix A.2 and Appendix A.3 with a simpler example. From a programming perspective [24], the correctness of these two views is established by the lightweight fusion program transformation [25].

7. From abstract machine to evaluation functions

This section implements the second half of the programme outlined in Section 4.9. We start from the big-step abstract machine of Figure 7 and refunctionalize it into a continuation-passing interpreter (Section 7.1), which we then map back to direct style (Section 7.2). Observing that a component of the resulting direct-style interpreter is in defunctionalized form, we refunctionalize it (Section 7.3). Refunctionalization and the direct-style transformation are illustrated in Appendix A.3, Appendix A.4 and Appendix A.5 with a simpler example.

7.1. Refunctionalization: from abstract machine to continuation-passing interpreter

Defunctionalization and refunctionalization:. Reynolds introduced defunctionalization [14, 26] to derive first-order evaluators from higher-order ones. Defunctionalization turns a

function type into a sum type, and function application into the application of an apply function dispatching on the sum type. Its left inverse, *refunctionalization* [13], can transform first-order abstract machines into higher-order interpreters. It specifically works on programs that are in defunctionalized form, i.e., in the image of Reynolds's defunctionalization.

Towards refunctionalizing the big-step abstract machine of Figure 7:. The big-step abstract machine of Figure 7 is not in defunctionalized form with respect to the inside-out evaluation contexts. Indeed these contexts are consumed by the two transition functions corresponding to $\langle E^{io}, A\rangle_{context}$ and $\langle E^{io}, (E^{oi}, x)\rangle_{reroot}$ rather than by the single apply function demanded for refunctionalization. This mismatch can be fixed by introducing a sum type discriminating between the (non-context) arguments to the two transition functions and combining them into a single transition function [13]. The left summand (tagged "*ans*") holds an answer, and the right summand (tagged "*var*") pairs a variable whose value is needed and an incrementally-constructed outside-in context used to get back to the place in the term where the value was needed.

Three of the context constructors occur on the right-hand sides of their own apply function clauses. When refunctionalized, these correspond to recursive functions and therefore appear as named functions.

The refunctionalized abstract machine is an interpreter for lazy evaluation in continuation-passing style, where the continuations are the functional representation of the inside-out contexts.

7.2. Back to direct style: from continuation-passing interpreter to natural semantics

It is a simple matter to transform the continuation-passing interpreter described in Section 7.1 into direct style [27]. The continuations do not represent any control effect other than non-tail calls, so the resulting direct-style interpreter does not require first-class control operators [28].

In the present case, the interpreter of Section 7.1 implements a natural semantics (i.e., a big-step operational semantics) for lazy evaluation. This semantics is displayed in Figure 8. In reference to Figure 7,

- there is one *term* transition and one $\Downarrow_{\text{eval}}$ judgement for each syntactic construct;
- for every *context* transition, there is a corresponding judgment over the *ans* injection tag:
  - two $\Downarrow_{\text{succ}}$ judgments for the two transitions on the frame "*succ* $\square$",
  - two $\Downarrow_{\text{apply}}$ judgments for the two transitions on the frame "$\square$ T",
  - one $\Downarrow_{\text{bind}}$ judgement for the transition on the frame "$\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ \square$", and
  - two $\Downarrow_{\text{force}}$ judgements for the two transitions on the frame "$\mathsf{let}\ x\ \mathsf{be}\ \square\ \mathsf{in}\ E^{oi}[x]$"; and
- for every *reroot* transition, there is a corresponding judgment over the *var* injection tag: one for each context frame, plus one for when there is a match.

$$\frac{}{\ulcorner n \urcorner\ {}^{X}{\Downarrow}^{X}_{\text{eval}}\ ans(\ulcorner n \urcorner)} \qquad \frac{T\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad r\ {}^{X'}{\Downarrow}^{X''}_{\text{succ}}\ r'}{succ\ T\ {}^{X}{\Downarrow}^{X''}_{\text{eval}}\ r'} \qquad \frac{}{x\ {}^{X}{\Downarrow}^{X}_{\text{eval}}\ var(x,\, \varepsilon^{oi})}$$

$$\frac{}{\lambda x.T\ {}^{X}{\Downarrow}^{X}_{\text{eval}}\ ans(\lambda x.T)} \qquad \frac{T_0\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad r\ T_1\ {}^{X'}{\Downarrow}^{X''}_{\text{apply}}\ r'}{T_0\ T_1\ {}^{X}{\Downarrow}^{X''}_{\text{eval}}\ r'}$$

$$\frac{T\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad (x,\, T_1,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T\ {}^{X}{\Downarrow}^{X''}_{\text{eval}}\ r'} \qquad \frac{}{ans(\ulcorner n \urcorner)\ {}^{X}{\Downarrow}^{X}_{\text{succ}}\ ans(\ulcorner n' \urcorner)}\text{where } n' = n + 1$$

$$\frac{ans(A)\ {}^{X}{\Downarrow}^{X'}_{\text{succ}}\ r \quad (x,\, T,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{ans(\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A)\ {}^{X}{\Downarrow}^{X''}_{\text{succ}}\ r'} \qquad \frac{}{var(x,\, E^{oi})\ {}^{X}{\Downarrow}^{X}_{\text{succ}}\ var(x,\, E^{oi} :: (succ\ \Box))}$$

$$\frac{T[x'/x]\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad (x',\, T_1,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{(ans(\lambda x.T))\ T_1\ {}^{(x',\, X)}{\Downarrow}^{X''}_{\text{apply}}\ r'} \qquad \frac{(ans(A))\ T_2\ {}^{X}{\Downarrow}^{X'}_{\text{apply}}\ r \quad (x,\, T_1,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{(ans(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A))\ T_2\ {}^{X}{\Downarrow}^{X''}_{\text{apply}}\ r'}$$

$$\frac{}{(var(x,\, E^{oi}))\ T_1\ {}^{X}{\Downarrow}^{X}_{\text{apply}}\ var(x,\, E^{oi} :: (\Box\ T_1))}$$

$$\frac{}{(x,\, T_1,\, ans(A))\ {}^{X}{\Downarrow}^{X}_{\text{bind}}\ ans(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A)} \qquad \frac{T_1\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad (x,\, r,\, E^{oi})\ {}^{X'}{\Downarrow}^{X''}_{\text{force}}\ r'}{(x,\, T_1,\, var(x,\, E^{oi}))\ {}^{X}{\Downarrow}^{X''}_{\text{bind}}\ r'}$$

$$\frac{}{(x,\, T_1,\, var(y,\, E^{oi}))\ {}^{X}{\Downarrow}^{X}_{\text{bind}}\ var(y,\, E^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ \Box))}\text{where } x \neq y$$

$$\frac{\langle V,\, E^{oi}\rangle_{oi} \Uparrow_{\text{rec}} T \quad T\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ r \quad (x,\, V,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{(x,\, ans(V),\, E^{oi})\ {}^{X}{\Downarrow}^{X''}_{\text{force}}\ r'}$$

$$\frac{(x,\, ans(A),\, E^{oi})\ {}^{X}{\Downarrow}^{X'}_{\text{force}}\ r \quad (y,\, T_1,\, r)\ {}^{X'}{\Downarrow}^{X''}_{\text{bind}}\ r'}{(x,\, ans(\mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A),\, E^{oi})\ {}^{X}{\Downarrow}^{X''}_{\text{force}}\ r'}$$

$$\frac{}{(x,\, var(y,\, E_1^{oi}),\, E^{oi})\ {}^{X}{\Downarrow}^{X}_{\text{force}}\ var(y,\, E_1^{oi} :: (\mathsf{let}\ x\ \mathsf{be}\ \Box\ \mathsf{in}\ E^{oi}[x]))}$$

Figure 8: Heapless natural semantics for call-by-need evaluation

Proposition 23 (full correctness)**.**

$$\langle T,\, \varepsilon^{io}\rangle_{term} \xrightarrow{X;X'\,*}_{\text{step}} \langle A\rangle_{answer} \quad\Leftrightarrow\quad T\ {}^{X}{\Downarrow}^{X'}_{\text{eval}}\ ans(A).$$

Proof. Corollary of the correctness of defunctionalization and the CPS transformation. □

As illustrated in Appendix A.6 and Appendix A.5, the natural semantics of Figure 8 is implemented as an interpreter in direct style. Following Reynolds's functional correspondence, it can be CPS transformed and defunctionalized towards the abstract machine of Figure 7.

$$
\begin{array}{rcl}
\mathsf{Result} & = & (\mu X.\mathsf{Answer} + (\mathsf{Var} \times (X \to X))) \times \mathsf{FreshVars} \\
\\
\text{eval} \ : \ \mathsf{Term} \times \mathsf{FreshVars} & \to & \mathsf{Result} \\
\text{eval}(\ulcorner n \urcorner,\, X) & = & (\mathit{ans}(\ulcorner n \urcorner),\, X) \\
\text{eval}(x,\, X) & = & (\mathit{var}(x,\, \overline{\lambda} r.r),\, X) \\
\text{eval}(\lambda x.T,\, X) & = & (\mathit{ans}(\lambda x.T),\, X) \\
\text{eval}(T_0\ T_1,\, X) & = & \text{apply}(\text{eval}(T_0,\, X),\, T_1) \\
\text{eval}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ T,\, X) & = & \text{bind}(x,\, T_1,\, \text{eval}(T,\, X)) \\
\text{eval}(\mathit{succ}\ T,\, X) & = & \text{succ}(\text{eval}(T,\, X)) \\
\\
\text{apply} \ : \ \mathsf{Result} \times \mathsf{Term} & \to & \mathsf{Result} \\
\text{apply}((\mathit{ans}(\lambda x.T),\, (x',\, X)),\, T_1) & = & \text{bind}(x',\, T_1,\, \text{eval}(T[x'/x],\, X)) \\
\text{apply}((\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A),\, X),\, T_2) & = & \text{bind}(x,\, T_1,\, \text{apply}((\mathit{ans}(A),\, X),\, T_2)) \\
\text{apply}((\mathit{var}(x,\, h),\, X),\, T_1) & = & (\mathit{var}(x,\, \overline{\lambda} r.\text{apply}(h \mathbin{\overline{@}} r,\, T_1)),\, X) \\
\\
\text{bind} \ : \ \mathsf{Var} \times \mathsf{Term} \times \mathsf{Result} & \to & \mathsf{Result} \\
\text{bind}(x,\, T_1,\, (\mathit{ans}(A),\, X)) & = & (\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T_1\ \mathsf{in}\ A),\, X) \\
\text{bind}(x,\, T_1,\, (\mathit{var}(x,\, h),\, X)) & = & \text{force}(x,\, \text{eval}(T_1,\, X),\, h) \\
\text{bind}(x,\, T_1,\, (\mathit{var}(y,\, h),\, X)) & = & (\mathit{var}(y,\, \overline{\lambda} r.\text{bind}(x,\, T_1,\, h \mathbin{\overline{@}} r)),\, X) \\
 & & \text{where } x \neq y \\
\\
\text{force} \ : \ \mathsf{Var} \times \mathsf{Result} \times (\mathsf{Result} \to \mathsf{Result}) & \to & \mathsf{Result} \\
\text{force}(x,\, (\mathit{ans}(V),\, X),\, h) & = & \text{bind}(x,\, V,\, h \mathbin{\overline{@}} (\mathit{ans}(V),\, X)) \\
\text{force}(x,\, (\mathit{ans}(\mathsf{let}\ y\ \mathsf{be}\ T_1\ \mathsf{in}\ A),\, X),\, h) & = & \text{bind}(y,\, T_1,\, \text{force}(x,\, (\mathit{ans}(A),\, X),\, h)) \\
\text{force}(x,\, (\mathit{var}(y,\, h'),\, X),\, h) & = & (\mathit{var}(y,\, \overline{\lambda} r.\text{force}(x,\, h' \mathbin{\overline{@}} r,\, h)),\, X) \\
\\
\text{succ} \ : \ \mathsf{Result} & \to & \mathsf{Result} \\
\text{succ}((\mathit{ans}(\ulcorner n \urcorner),\, X)) & = & (\mathit{ans}(\ulcorner n' \urcorner),\, X) \quad \text{where } n' = n + 1 \\
\text{succ}((\mathit{ans}(\mathsf{let}\ x\ \mathsf{be}\ T\ \mathsf{in}\ A),\, X)) & = & \text{bind}(x,\, T,\, \text{succ}((\mathit{ans}(A),\, X))) \\
\text{succ}((\mathit{var}(x,\, h),\, X)) & = & (\mathit{var}(x,\, \overline{\lambda} r.\text{succ}(h \mathbin{\overline{@}} r)),\, X)
\end{array}
$$

Figure 9: The heapless natural semantics of Figure 8 after refunctionalization

7.3. Refunctionalization: from natural semantics to higher-order evaluation function

The natural-semantics implementation of Section 7.2 is already in defunctionalized form with respect to the first-order outside-in contexts. Indeed, as already mentioned in Section 4.4, the recomposition function of Definition 4 and Figure 1 is the corresponding apply function.

An outside-in context acts as an accumulator recording the path from a variable whose value is needed to its binding site. The recomposition function turns this accumulator inside-out again when the variable's value is found. The refunctionalized outside-in contexts are functional representations of these accumulators.

The resulting refunctionalized evaluation function is displayed in Figure 9. Notationally, higher-order functions are introduced with $\overline{\lambda}$ and eliminated with $\overline{@}$, which is infix.

Proposition 24 (full correctness)**.**

$$T \; {}^{X}\!\Downarrow^{X'}_{\text{eval}} \; ans(A) \quad \Leftrightarrow \quad \text{eval}(T,\, X) = (ans(A),\, X').$$

Proof. Corollary of the correctness of defunctionalization. □

This higher-order evaluation function exhibits a computational pattern that we find striking because it also occurs in Cartwright and Felleisen's work on extensible denotational language specifications [29]: each valuation function yields either a (left-injected with "*ans*") value or a (right-injected with "*var*") variable together with a higher-order function. For each call, this higher-order function may yield another right-injected higher-order function that, when applied, restores this current call. As illustrated in Appendix B, this computational pattern is typical of control: the left inject stands for an expected result, while the right inject acts as an exceptional return that incrementally captures the current continuation. This observation also points at a structural commonality in Ariola and Felleisen's small-step semantics [1], which uses delimited control, and in Cartwright and Felleisen's big-step semantics [29], which uses undelimited control. At any rate, for undelimited control, this computational pattern was subsequently re-invented by Fünfrocken to implement process migration [30–32], and then put to use to implement first-class continuations [33, 34]. In the present case, this pattern embodies two distinct computational aspects—one intensional and the other extensional:

How: The computational pattern is one of delimited control, from the point of use of a let-declared variable to its point of declaration.

What: The computational effect is one of a write-once state since once the delimited context is captured, it is restored with the value of the let-declared variable.

These two aspects were instrumental in Cartwright and Felleisen's design of extensible denotational semantics for (undelimited) Control Scheme and for State Scheme [29]. For let insertion in partial evaluation, these control and state aspects were re-discovered and put to use by Sumii and Kobayashi [35], and for let insertion in type-directed partial evaluation, by Grobauer and Yang [36]. For normalization by evaluation, this control aspect was also re-discovered and put to use by Balat et al. [37], who abstract delimited control from the use site of a lambda-declared variable to its definition site. For call by need, this control aspect was recently identified and put to new use by Garcia, Lumsdaine and Sabry [4], and this store aspect was originally envisioned by Vuillemin [38], Wadsworth [39], and initially Landin [40].

These observations put us in position to write the evaluation function of Figure 9 in direct style, either with delimited control operators (one control delimiter for each let declaration, and one control abstraction for each occurrence of a let-declared variable whose value is needed), or with a state monad, as illustrated in Appendix B with a simpler example.

8. Deterministic abstract machines define functions

Up to Section 6, we scrupulously described small-step computation with relations, before shifting to functions for describing big-step computation. However, for deterministic programming languages, functions are sufficient to describe small-step computation,

as done throughout the first author's lecture notes at AFP 2008 [15]. For example, in the present work, the decomposition and recomposition functions of Section 4, together with the data type of contexts, are in defunctionalized form. They can therefore easily be refunctionalized, as illustrated in Appendix A. This representational flexibility indicates a large and friendly degree of expressive freedom for implementing reduction semantics and one-step reduction functions in a functional programming language, not just for the call-by-need λ-calculus, but in general.

9. Conclusion

Semantics should be call by need.
– Rod Burstall

Over the years, the two key features of lazy evaluation – demand-driven computation and memoization of intermediate results – have elicited a fascinating variety of semantic artifacts, each with its own originality and elegance. It is our overarching thesis that spelling out the methodical search for the next potential redex that is implicit in a reduction semantics paves the way towards other semantic artifacts that not only are uniformly inter-derivable and sound by construction *but also match what a programming-language semanticist crafts by hand.* Elsewhere, we have already shown that refocusing, etc. do not merely apply to purely syntactic theories such as, e.g., Felleisen and Hieb's syntactic theories of sequential control and state [41, 42]: the methodology also applies to call by need with a global heap of memo-thunks [6, 20, 43] and to combinatory graph reduction, connecting term graph rewriting systems à la Barendregt et al. and graph-reduction machines à la Turner [44, 45]. Here, we have shown that the methodology also applies to Ariola et al.'s purely syntactic account of call by need.

Acknowledgments:. Thanks are due to the anonymous reviewers. We are also grateful to Zena Ariola, Kenichi Asai, Ronald Garcia, Oleg Kiselyov, Kristoffer Rose, Ilya Sergey and Chung-chieh Shan for discussions and comments.

The first author heard Rod Burstall's quote in Section 9 from Neil Jones in the late 1980s, but was unable to locate it in writing. In May 2009, he asked Rod Burstall about it: Rod Burstall made that statement at Edinburgh at the occasion of a seminar by Christopher Wadsworth in the course of the 1970s.

Appendix A. On refunctionalizing and going back to direct style

The goal of this appendix is to illustrate refunctionalization and the direct-style transformation. Our running example is an evaluator for a simple language of arithmetic expressions.

Appendix A.1. Abstract machine for evaluating arithmetic expressions

Our language of arithmetic expressions reads as follows:

$$\begin{aligned} \textsf{Term} \ni T &::= \ulcorner n \urcorner \mid T + T \mid T \times T \\ \textsf{Value} \ni V &::= \ulcorner n \urcorner \\ \textsf{Evaluation Context} \ni E &::= [\,] \mid E + T \mid V + E \mid E \times T \mid V \times E \end{aligned}$$

In words: terms are integers, additions, and multiplications; values are integers; and evaluation contexts specify a left-most inner-most reduction order.

Here is an abstract machine for this language:

$$
\begin{array}{rcl}
\langle \ulcorner n \urcorner,\, E\rangle_{term} & \rightarrow_{\text{step}} & \langle E,\, \ulcorner n \urcorner\rangle_{context} \\
\langle T_1 + T_2,\, E\rangle_{term} & \rightarrow_{\text{step}} & \langle T_1,\, E + T_2\rangle_{term} \\
\langle T_1 \times T_2,\, E\rangle_{term} & \rightarrow_{\text{step}} & \langle T_1,\, E \times T_2\rangle_{term} \\
\\
\langle [\,],\, \ulcorner n \urcorner\rangle_{context} & \rightarrow_{\text{step}} & \langle \ulcorner n \urcorner\rangle_{value} \\
\langle E + T_2,\, \ulcorner n_1 \urcorner\rangle_{context} & \rightarrow_{\text{step}} & \langle T_2,\, \ulcorner n_1 \urcorner + E\rangle_{term} \\
\langle \ulcorner n_1 \urcorner + E,\, \ulcorner n_2 \urcorner\rangle_{context} & \rightarrow_{\text{step}} & \langle E,\, \ulcorner n \urcorner\rangle_{context} \\
 & & \text{where } n = n_1 + n_2 \\
\langle E \times T_2,\, \ulcorner n_1 \urcorner\rangle_{context} & \rightarrow_{\text{step}} & \langle T_2,\, \ulcorner n_1 \urcorner \times E\rangle_{term} \\
\langle \ulcorner n_1 \urcorner \times E,\, \ulcorner n_2 \urcorner\rangle_{context} & \rightarrow_{\text{step}} & \langle E,\, \ulcorner n \urcorner\rangle_{context} \\
 & & \text{where } n = n_1 \times n_2
\end{array}
$$

This abstract machine consists of three states: the first defines the relation on terms, the second defines the relation on evaluation contexts, and the third is the terminal state of values. A term T evaluates to a value V iff $\langle T,\, [\,]\rangle_{term} \rightarrow^*_{\text{step}} \langle V\rangle_{value}$.

Appendix A.2. Small-step implementation of the abstract machine

The abstract machine of Section Appendix A.1 can be regarded as a small-step abstract machine defining a relation between any non-final state and its successive state. Let us implement it in Haskell.

Terms, values and evaluation contexts read as follows:

```
data Term = Num Int | Add Term Term | Mul Term Term
type Val  = Int
data Cont = Empty
          | EAddL Cont Term | EAddR Val Cont
          | EMulL Cont Term | EMulR Val Cont
```

States are represented with a data type:

```
data State = TERM Term Cont | CONT Cont Val | VAL Val
```

The `TERM` constructor is used to represent *term* states, `CONT` to represent *context* states, and `VAL` to represent the final state.

Transitions are implemented with a function associating each non-final state to its successive state:

```
step :: State → State
step (TERM (Num n)       e) = CONT e n
step (TERM (Add t1 t2)   e) = TERM t1 (EAddL e t2)
step (TERM (Mul t1 t2)   e) = TERM t1 (EMulL e t2)
step (CONT Empty         n) = VAL n
step (CONT (EAddL e t2) n1) = TERM t2 (EAddR n1 e)
step (CONT (EAddR n1 e) n2) = CONT e (n1 + n2)
step (CONT (EMulL e t2) n1) = TERM t2 (EMulR n1 e)
step (CONT (EMulR n1 e) n2) = CONT e (n1 * n2)
```

Evaluating a term is done by starting in the initial *term* state with the term and the empty context and iterating the transition sequence towards a final state:

```
iterate :: State → Val
iterate (VAL n) = n
iterate state   = iterate (step state)

main0 :: Term → Val
main0 t = iterate (TERM t Empty)
```

Appendix A.3. Big-step implementation of the abstract machine

The abstract machine of Section Appendix A.1 can be equally regarded as a big-step abstract machine defining a function from terms to values [24]. Let us implement it in Haskell.

Terms, values and evaluation contexts read as in Section Appendix A.2.

Transitions are implemented with a set of mutually tail-recursive functions:

```
term :: Term → Cont → Val
term (Num n)     e = cont e n
term (Add t1 t2) e = term t1 (EAddL e t2)
term (Mul t1 t2) e = term t1 (EMulL e t2)

cont :: Cont → Val → Val
cont Empty         n  = n
cont (EAddL e t2) n1 = term t2 (EAddR n1 e)
cont (EAddR n1 e) n2 = cont e (n1 + n2)
cont (EMulL e t2) n1 = term t2 (EMulR n1 e)
cont (EMulR n1 e) n2 = cont e (n1 * n2)

main1 :: Term → Val
main1 t = term t Empty
```

The `term` function represents transitions from *term* states; the `cont` function represents transitions from *context* states; and the final return value represents the final *value* states. Evaluating a term is done by invoking the `term`-transition with the term and the empty context.

This implementation is in defunctionalized form with respect to the data type of evaluation contexts, `Cont`, and the function, `cont`, dispatching on that data type: each data constructor of `Cont` is consumed by `cont` which implements how to continue evaluation. Refunctionalization replaces each call to a data constructor of `Cont` by the introduction of a function that implements how to continue evaluation, and each call to `cont` by the elimination of this function, i.e., its application. For example, `cont` maps the data constructor `Empty` to the identity function; `Empty` is thus refunctionalized as the identity function. The function implementing how to continue evaluation is of course the continuation of an evaluator.

Appendix A.4. Continuation-passing evaluator

Refunctionalizating the abstract machine of Section Appendix A.3 yields the following evaluator, which is in continuation-passing style (CPS):

```
evalc :: Term → (Val → a) → a
evalc (Num n)     k = k n
evalc (Add t1 t2) k = evalc t1 (λn1 → evalc t2 (λn2 → k (n1 + n2)))
evalc (Mul t1 t2) k = evalc t1 (λn1 → evalc t2 (λn2 → k (n1 * n2)))

main2 :: Term → Val
main2 t = evalc t (λn → n)
```

This evaluator is in CPS since all calls are tail calls and the second parameter is a continuation.

Appendix A.5. Direct-style evaluator

Applying the direct-style transformation, i.e., the left inverse of the CPS transformation [27], to the continuation-passing evaluator of Section Appendix A.4, we obtain the following evaluator, which is in direct style:

```
eval :: Term → Val
eval (Num n)      = n
eval (Add t1 t2) = eval t1 + eval t2
eval (Mul t1 t2) = eval t1 * eval t2

main3 :: Term → Val
main3 t = eval t
```

CPS-transforming this direct-style evaluator yields the continuation-passing evaluator of Section Appendix A.4. Defunctionalizing this continuation-passing evaluator yields the abstract machine of Section Appendix A.3. This sequence of program transformations was introduced in Reynolds's work on definitional interpreters 4 decades ago [26]. It was put in the limelight, together with the converse sequence, in the past decade [11, 46].

Appendix A.6. Natural semantics

The direct-style interpreter of Section Appendix A.5 implements the following (big-step) natural semantics:

$$\frac{}{\ulcorner n \urcorner \Downarrow_{\text{eval}} \ulcorner n \urcorner} \qquad \frac{T_1 \Downarrow_{\text{eval}} \ulcorner n_1 \urcorner \quad T_2 \Downarrow_{\text{eval}} \ulcorner n_2 \urcorner}{T_1 + T_2 \Downarrow_{\text{eval}} \ulcorner n \urcorner} \text{ where } n = n_1 + n_2$$

$$\frac{T_1 \Downarrow_{\text{eval}} \ulcorner n_1 \urcorner \quad T_2 \Downarrow_{\text{eval}} \ulcorner n_2 \urcorner}{T_1 \times T_2 \Downarrow_{\text{eval}} \ulcorner n \urcorner} \text{ where } n = n_1 \times n_2$$

A term T evaluates to a value V iff $T \Downarrow_{\text{eval}} V$.

The present natural semantics and the abstract machine of Section Appendix A.1 are thus uniformly inter-derivable, and they match what a programming-language semanticist would craft by hand (see Footnote 4, page 16 for a non-trivial recent example).

Appendix B. On the control pattern underlying call by need

The goal of this appendix is to illustrate the control pattern of Figure 9. Our running example counts the number of occurrences of each bound variable in a λ-term. More precisely, we define a function mapping a closed λ-term of type `Term1` into a new λ-term of type `Term2` where each binder $\lambda x.T$ has been tagged with the number of free occurrences of x in T.

```
data Term1 = Var1 String
           | Lam1 String Term1
           | App1 Term1 Term1

data Term2 = Var2 String
           | Lam2 String Int Term2
           | App2 Term2 Term2
```

We present three definitions: one with the control pattern of Figure 9 (Appendix Appendix B.1), one with its direct-style counterpart using control operators (Appendix Appendix B.2), and one in state-passing style (Appendix Appendix B.3). All three are implemented in Haskell. We have tested them with the Glasgow Haskell Compiler.

Appendix B.1. Function-based encoding

The main function, `count`$_1$, calls an auxiliary function, `visit`, that returns the characteristic sum type of intermediate results: the current answer, left-injected with "`Ans`", or a function resuming the computation of the current intermediate answer, right-injected with "`Var`" and tagged with the variable under consideration:

```
data Intermediate
  = Ans Term2
  | Var String (() → Intermediate)

count₁ t = case visit t of
    Ans t'  → t'
    Var x h → error "open term"
  where
    visit :: Term1 → Intermediate
    visit (Var1 x)      = var x
    visit (Lam1 x t)    = lam x 0 (visit t)
    visit (App1 t0 t1) = app (visit t0) (visit t1)

    var x = Var x (λ() → Ans (Var2 x))

    lam x n (Ans t) = Ans (Lam2 x n t)
    lam x n (Var y h)
      | x == y       = lam x (n + 1) (h ())
      | otherwise    = Var y (lam x n ∘ h)

    app (Ans t0) (Ans t1) = Ans (App2 t0 t1)
    app (Var x h) r = Var x ((λs → app s r) ∘ h)
    app r (Var x h) = Var x ((λs → app r s) ∘ h)
```

Each time a variable is visited, its continuation is captured from its point of use to its point of definition, its count is incremented, and the captured continuation is restored. The capture is realized by bubbling up with `Var`, as it were, from a point of use to its lexical point of definition while accumulating a delimited continuation by function composition. The restoration is realized by applying this delimited continuation.

Appendix B.2. Continuation-based encoding

The main function, `count`$_2$, calls an auxiliary function, `visit`, that delimits control for each variable definition, and abstracts control for each variable use, using Dybvig, Peyton-Jones and Sabry's monadic framework for subcontinuations [47]:

```
import Control.Monad.CC

count₂ t = runCC (visit t [])
  where
    visit :: MonadDelimitedCont p s m ⇒
             Term1 → [(String, Term2 → m Term2)] → m Term2
    visit (Var1 x) ms =
      (mark x ms) (Var2 x)
```

```
    visit (Lam1 x t) ms = do
      h ← reset (λp → do
        let k t = shift p (λk → do
                    h ← k (return t)
                    return (λn → h (n + 1)))
        t' ← visit t ((x, k) : ms)
        shift p (λk → return (λn → Lam2 x n t')))
      return (h 0)
    visit (App1 t1 t2) ms = do
      t1' ← visit t1 ms
      t2' ← visit t2 ms
      return (App2 t1' t2')

    mark x ms =
      case lookup x ms of
        Just p  → p
        Nothing → error "open term"
```

This implementation reflects the control pattern in Appendix B.1 in that the computation incrementing the counter is defined at the point of variable definition. However, since the control abstraction is a closed term here, we can unfold it at the point of variable usage:

```
count₃ t = runCC (visit t [])
  where
    visit :: MonadDelimitedCont p s m ⇒
             Term1 → [(String, p (Int → Term2))] → m Term2
    visit (Var1 x) ms =
      shift (mark x ms) (λk → do
        h ← k (return (Var2 x))
        return (λn → h (n + 1)))
    visit (Lam1 x t) ms = do
      h ← reset (λp → do
        t' ← visit t ((x, p) : ms)
        shift p (λk → return (λn → Lam2 x n t')))
      return (h 0)
    visit (App1 t0 t1) ms = do
      t0' ← visit t0 ms
      t1' ← visit t1 ms
      return (App2 t0' t1')

    mark x ms =
      case lookup x ms of
        Just p  → p
        Nothing → error "open term"
```

Each time a variable is visited, its continuation is captured from its point of use to its point of definition, its count is incremented, and the captured continuation is restored. The capture is realized by using the control operator `shift`, which abstracts control into a delimited continuation. The restoration is realized by applying this delimited continuation.

Appendix B.3. State-based encoding

The main function, $\texttt{count}_4$, calls an auxiliary function, `visit`, that threads an association list of declared variables and numbers of their occurrences by use of a state monad:

```
import Control.Monad.State

count4 t = evalState (visit t) []
  where
    visit :: Term1 → State [(String, Int)] Term2
    visit (Var1 x) = do
      modify (incr x)
      return (Var2 x)
    visit (Lam1 x t) = do
      modify ((x, 0):)
      t' ← visit t
      n  ← gets (snd ∘ head)
      modify tail
      return (Lam2 x n t')
    visit (App1 t0 t1) = do
      t0' ← visit t0
      t1' ← visit t1
      return (App2 t0' t1')

    incr x [] = error "open term"
    incr x ((y, n) : ys)
      | x == y    = (y, n + 1) : ys
      | otherwise = (y, n    ) : incr x ys
```

Each time a variable is visited, its association is accessed in the current state, the count in this association is incremented, which yields a new state, and the computation is resumed in this new state.

References

[1] Z. M. Ariola, M. Felleisen, The call-by-need lambda calculus, Journal of Functional Programming 7 (3) (1997) 265–301.

[2] J. Maraist, M. Odersky, P. Wadler, The call-by-need lambda calculus, Journal of Functional Programming 8 (3) (1998) 275–317.

[3] D. P. Friedman, A. Ghuloum, J. G. Siek, L. Winebarger, Improving the lazy Krivine machine, Higher-Order and Symbolic Computation 20 (3) (2007) 271–293.

[4] R. Garcia, A. Lumsdaine, A. Sabry, Lazy evaluation and delimited control, Logical Methods in Computer Science 6 (3:1) (2010) 1–39, a preliminary version was presented at the Thirty-Sixth Annual ACM Symposium on Principles of Programming Languages (POPL 2009).

[5] P. Sestoft, Deriving a lazy abstract machine, Journal of Functional Programming 7 (3) (1997) 231–264.

[6] M. S. Ager, O. Danvy, J. Midtgaard, A functional correspondence between call-by-need evaluators and lazy abstract machines, Information Processing Letters 90 (5) (2004) 223–232, extended version available as the research report BRICS RS-04-3.

[7] J. Launchbury, A natural semantics for lazy evaluation, in: S. L. Graham (Ed.), Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, ACM Press, Charleston, South Carolina, 1993, pp. 144–154.

[8] P. Henderson, J. H. Morris Jr., A lazy evaluator, in: S. L. Graham (Ed.), Proceedings of the Third Annual ACM Symposium on Principles of Programming Languages, ACM Press, 1976, pp. 95–103.

[9] M. B. Josephs, The semantics of lazy functional languages, Theoretical Computer Science 68 (1989) 105–111.

[10] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, revised Edition, Vol. 103 of Studies in Logic and the Foundation of Mathematics, North-Holland, 1984.

[11] O. Danvy, Defunctionalized interpreters for programming languages, in: J. Hook, P. Thiemann (Eds.), Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08), SIGPLAN Notices, Vol. 43, No. 9, ACM Press, Victoria, British Columbia, 2008, pp. 131–142, invited talk.

[12] O. Danvy, L. R. Nielsen, Refocusing in reduction semantics, Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, Aarhus, Denmark, a preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol. 59.4 (Nov. 2004).

[13] O. Danvy, K. Millikin, Refunctionalization at work, Science of Computer Programming 74 (8) (2009) 534–549, extended version available as the research report BRICS RS-08-04.

[14] O. Danvy, L. R. Nielsen, Defunctionalization at work, in: H. Søndergaard (Ed.), Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01), ACM Press, Firenze, Italy, 2001, pp. 162–174, extended version available as the research report BRICS RS-01-23; most influential paper at PPDP 2001.

[15] O. Danvy, From reduction-based to reduction-free normalization, in: P. Koopman, R. Plasmeijer, D. Swierstra (Eds.), Advanced Functional Programming, Sixth International School, no. 5382 in Lecture Notes in Computer Science, Springer, Nijmegen, The Netherlands, 2008, pp. 66–164, lecture notes including 70+ exercises.

[16] A. Bondorf, O. Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, Science of Computer Programming 16 (1991) 151–195.

[17] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, P. Wadler, A call-by-need lambda calculus, in: P. Lee (Ed.), Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages, ACM Press, San Francisco, California, 1995, pp. 233–246.

[18] R. Bloo, K. H. Rose, Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection, in: CSN-95: Computer Science in the Netherlands, 1995, pp. 62–72.

[19] G. Huet, The zipper, Journal of Functional Programming 7 (5) (1997) 549–554.

[20] M. Pirog, D. Biernacki, A systematic derivation of the STG machine verified in Coq, in: J. Gibbons (Ed.), Haskell '10: Proceedings of the 2010 ACM SIGPLAN Haskell Symposium, ACM Press, Baltimore, Maryland, 2010, pp. 25–36.

[21] S. L. Peyton Jones, Implementing lazy functional languages on stock hardware: The spineless tagless G-machine, Journal of Functional Programming 2 (2) (1992) 127–202.

[22] M. Felleisen, M. Flatt, Programming languages and lambda calculi, unpublished lecture notes available at <http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html> and last accessed in April 2008 (1989-2001).

[23] F. Sieczkowski, M. Biernacka, D. Biernacki, Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in Coq, in: 22nd Symposium on Implementation and Application of Functional Languages, (IFL'10), Lecture Notes in Computer Science, Springer, Alphen aan den Rijn, The Netherlands, 2010, to appear.

[24] O. Danvy, K. Millikin, On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion, Information Processing Letters 106 (3) (2008) 100–109.

[25] A. Ohori, I. Sasano, Lightweight fusion by fixed point promotion, in: M. Felleisen (Ed.), Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages, SIGPLAN Notices, Vol. 42, No. 1, ACM Press, Nice, France, 2007, pp. 143–154.

[26] J. C. Reynolds, Definitional interpreters for higher-order programming languages, in: Proceedings of 25th ACM National Conference, Boston, Massachusetts, 1972, pp. 717–740, reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword [48].

[27] O. Danvy, Back to direct style, Science of Computer Programming 22 (3) (1994) 183–195, a preliminary version was presented at the Fourth European Symposium on Programming (ESOP 1992).

[28] O. Danvy, J. L. Lawall, Back to direct style II: First-class continuations, in: W. Clinger (Ed.), Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, LISP Pointers, Vol. V, No. 1, ACM Press, San Francisco, California, 1992, pp. 299–310.

[29] R. Cartwright, M. Felleisen, Extensible denotational language specifications, in: M. Hagiya, J. C. Mitchell (Eds.), Proceedings of the 1994 International Symposium on Theoretical Aspects of Computer Software, no. 789 in Lecture Notes in Computer Science, Springer-Verlag, Sendai, Japan, 1994, pp. 244–272.

[30] S. Fünfrocken, Transparent migration of Java-based mobile agents, in: K. Rothermel, F. Hohl (Eds.), Mobile Agents, Second International Workshop, MA'98, Proceedings, Vol. 1477 of Lecture Notes in Computer Science, Springer, Stuttgart, Germany, 1998, pp. 26–37.

[31] T. Sekiguchi, H. Masuhara, A. Yonezawa, A simple extension of Java language for controllable transparent migration and its portable implementation, in: P. Ciancarini, A. L. Wolf (Eds.), Coordination Languages and Models, Third International Conference, COORDINATION '99, Proceedings, Vol. 1594 of Lecture Notes in Computer Science, Springer, Amsterdam, The Netherlands,

1999, pp. 211–226.
[32] W. Tao, A portable mechanism for thread persistence and migration, Ph.D. thesis, University of Utah, Salt Lake City, Utah (2001).
[33] F. Loitsch, Scheme to JavaScript compilation, Ph.D. thesis, Université de Nice, Nice, France (Mar. 2009).
[34] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, M. Felleisen, Continuations from generalized stack inspection, in: O. Danvy, B. C. Pierce (Eds.), Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming (ICFP'05), SIGPLAN Notices, Vol. 40, No. 9, ACM Press, Tallinn, Estonia, 2005, pp. 216–227.
[35] E. Sumii, N. Kobayashi, A hybrid approach to online and offline partial evaluation, Higher-Order and Symbolic Computation 14 (2/3) (2001) 101–142.
[36] B. Grobauer, Z. Yang, The second Futamura projection for type-directed partial evaluation, Higher-Order and Symbolic Computation 14 (2/3) (2001) 173–219.
[37] V. Balat, R. D. Cosmo, M. P. Fiore, Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums, in: X. Leroy (Ed.), Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages, SIGPLAN Notices, Vol. 39, No. 1, ACM Press, Venice, Italy, 2004, pp. 64–76.
[38] J. Vuillemin, Correct and optimal implementations of recursion in a simple programming language, Journal of Computer and System Sciences 9 (3) (1974) 332–354.
[39] C. P. Wadsworth, Semantics and pragmatics of the lambda calculus, Ph.D. thesis, Computing Laboratory, Oxford University, Oxford, UK (1971).
[40] P. J. Landin, The mechanical evaluation of expressions, The Computer Journal 6 (4) (1964) 308–320.
[41] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical Computer Science 103 (2) (1992) 235–271.
[42] J. Munk, A study of syntactic and semantic artifacts and its application to lambda definability, strong normalization, and weak normalization in the presence of state, Master's thesis, Department of Computer Science, Aarhus University, Aarhus, Denmark, bRICS research report RS-08-3 (May 2007).
[43] M. Biernacka, O. Danvy, A syntactic correspondence between context-sensitive calculi and abstract machines, Theoretical Computer Science 375 (1-3) (2007) 76–108, extended version available as the research report BRICS RS-06-18.
[44] O. Danvy, I. Zerny, Three syntactic theories for combinatory graph reduction, in: M. Alpuente (Ed.), Logic Based Program Synthesis and Transformation, 20th International Symposium, LOPSTR 2010, revised selected papers, no. 6564 in Lecture Notes in Computer Science, Springer, Hagenberg, Austria, 2010, pp. 1–20, invited talk.
[45] I. Zerny, On graph rewriting, reduction and evaluation, in: Z. Horváth, V. Zsók, P. Achten, P. Koopman (Eds.), Trends in Functional Programming, Volume 10, Intellect Books, Komárno, Slovakia, 2009, pp. 81–112, granted the best student-paper award of TFP 2009.
[46] M. S. Ager, D. Biernacki, O. Danvy, J. Midtgaard, A functional correspondence between evaluators and abstract machines, in: D. Miller (Ed.), Proceedings of the Fifth ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03), ACM Press, Uppsala, Sweden, 2003, pp. 8–19.
[47] R. K. Dybvig, S. Peyton-Jones, A. Sabry, A monadic framework for subcontinuations, Journal of Functional Programming 17 (6) (2007) 687–730.
[48] J. C. Reynolds, Definitional interpreters revisited, Higher-Order and Symbolic Computation 11 (4) (1998) 355–361.